

Chapter 15

Parallel-NetCDF

Rob Latham

Argonne National Laboratory

15.1	Motivation	179
15.2	History and Background	180
15.3	Design and Architecture	181
15.4	Deployment and Usage	182
15.5	Additional Features	183
15.6	Conclusion	184
15.7	Additional Resources	185
	Bibliography	186

15.1 Motivation

Parallel-NetCDF (often abbreviated as simply “pNetCDF”), is a high-level parallel I/O library providing a portable, self-describing file format for the storage of multi-dimensional arrays of typed data. Parallel-NetCDF provides an application-oriented interface on top of the more general, lower level MPI-IO interface, while maintaining the standard netCDF file format.

Computational scientists, in addition to mastering their scientific domain, also have the challenge of mastering the computer systems upon which their simulations run. To help hide the hardware details of these machines, scientists can use compilers, programming languages, and math kernel libraries. In order to manage the tens of thousands of nodes and millions of processing elements, scientists can rely on communication middleware such as MPI. In much the same way, scientists have at their disposal *high-level I/O libraries*. These I/O libraries sit atop an I/O middleware (such as MPI-IO, discussed in Chapter 13), packaging and presenting some of the complexity of the lower-level, general purpose middleware into a format more appropriate for a scientific programmer. Figure 15.1 depicts this I/O software stack, and Parallel-NetCDF’s role in it.

The “data problem” facing computational scientists consists of three main challenges. First, the simulations of physical phenomena improve their fidelity with each new generation of hardware. As the gap between what computer

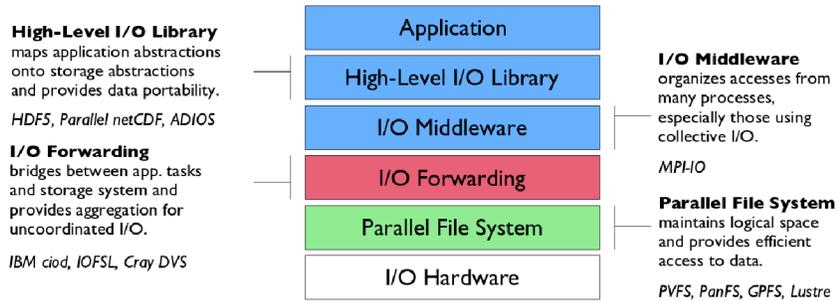


FIGURE 15.1: Parallel-NetCDF sits just below applications in the I/O software stack. It provides a more application-oriented interface to the more general and complex MPI-IO library. Applications can describe I/O needs in terms of multi-dimensional arrays. Parallel-NetCDF turns these requests into MPI-IO collective I/O operations.

simulations predict and reality bears out shrinks, the amount of data produced by these simulations grows. In 2012, DOE INCITE applications for time on high-end computing platforms routinely predicted needing terabytes of data. Second, hard drives double in capacity every 18 months, but the performance of a hard drive does not match that pace. In order to achieve high storage rates, many devices must be harnessed in parallel. The computational scientist looking to produce or analyze terabytes of data needs some way to manage parallelism in the I/O layer.

Parallelism in the I/O layer brings its own challenges. In order to provide high performance, storage systems deploy a large number of disks, servers, and storage links. Applications could operate directly upon these parallel storage systems. For the sake of developer productivity and I/O performance, however, I/O libraries exist to provide the abstractions and optimizations computational scientists need.

Stepping back from the topic of performance briefly and taking a broader view of the role of high-level I/O libraries in the life of a modern computational scientist, scientists operate in collaborations, exchanging datasets with other scientists conducting experiments on machines with different architectural characteristics. Raw binary files or custom file formats complicate the collaboration story. Instead, standard file formats mean every scientist can read data produced by any other scientists. Portability in file formats mean no matter the byte-endianess or word size of a machine, the data will always look the same. One can also imagine “portability over time”: these I/O libraries provide routines to annotate the stored data, or the entire file itself. These annotations help put the data in context, describing when the data was produced, by what application, or something as mundane yet important as what units the data are in.

15.2 History and Background

In 2002, when Argonne National Laboratory and Northwestern University started the Parallel-NetCDF project, the climate community had for nearly a decade prior been using the serial netCDF package [6] from UCAR. Serial netcdf provided climate scientists half of what they needed: the library and file format had at its foundation the kinds of multi-dimensional arrays of typed data that naturally fit with the kinds of simulations climate scientists carry out.

The missing half of serial netCDF was how to access these datasets in parallel. At the time, simulations faced two unappealing choices. Either they could do “file-per-process” I/O, producing one netCDF file for each parallel process, or they could send all data to a master process and have that process do all I/O. An “N-to-N” I/O model, where N processes operate on N (or more) files, quickly poses challenges to the underlying file system as it tries to deal with thousands of files. Writing N-to-N is far simpler than reading N files and re-assembling the simulation state. Sending a collection of N files to a collaborator also poses challenges. A far better solution would be to just operate on one file.

Sending all data to a master rank to manage one file poses two challenges. First, the master process needs enough memory to hold data from the other parallel processors. Second, the master process quickly becomes a critical resource, preventing all other processes from making progress. In an era where thousand-way parallelism is routine, an approach that serializes access to one processor may certainly be possible, but will result in unacceptable bottlenecks.

At the time, the only other application-oriented I/O library was HDF5. Like netCDF, HDF5 provided (and continues to provide) a data model and API well-suited to multi-dimensional arrays of typed data. (See Chapter 16 for more information about HDF5.) The HDF5 API and model differs significantly from netCDF’s API and file format. Those HDF5 differences allow for many powerful features, but make some optimization more difficult. A parallel version of netCDF offered a chance to explore parallel I/O in a more constrained context.

15.3 Design and Architecture

The Parallel-NetCDF design should look familiar to anyone familiar with serial netCDF. Having existed a decade prior to Parallel-NetCDF, serial netCDF had already established data files and codes. Nothing about the es-

HEADER	VARIABLE	VARIABLE	R1	R2	R1	R2	R1	R2
--------	----------	----------	----	----	----	----	----	----

FIGURE 15.2: The netCDF file layout (used in both serial netCDF and Parallel-NetCDF). A small header describes the contents of the file and any attributes on variables, dimensions, or the file itself. Variables follow in contiguous regions. The “non-record variables,” those that can grow in one dimension, follow in an interleaved fashion.

established file format precluded parallel I/O, so Parallel-NetCDF used that file format as the foundation. By maintaining file format compatibility, scientists could introduce Parallel-NetCDF to their workflows with minimal disruption. Parallel-NetCDF could not implement some of its ideas for optimized parallel I/O without altering the programming API, but such alterations were done in the spirit of serial-NetCDF, and retain much of the same feel and semantics.

Both libraries support the “classic” netCDF file format, depicted in Figure 15.2. The first bytes of the file contain a header. This header describes the name and size of the dimensions used by the variables; the name, type, shape, and starting location in the file of variables; and annotations or “attributes” associated with dimensions, variables, or the entire dataset.

Serial netCDF introduced the idea of a bi-modal API: to write a dataset, a program enters “define mode.” In define mode, the programmer defines the dimensions being used, associates those dimensions with variables, possibly annotates the components of the dataset, and then switches to “data mode.” The program can only transfer data after it has described how the data will be stored. This pre-declaration of the file structure benefits parallel I/O greatly: a process can compute where every element of every array will go, and need not coordinate with any other process. The listing in Figure 15.3 provides a brief example of how define mode and data mode work together. Readers can find entire code samples in the on-line Parallel-Netcdf Quick Tutorial [1].

15.4 Deployment and Usage

In many situations, the choice of which I/O library to use depends on the domain in which one operates. Climate codes, for example, have a large repository of dataset in the classic netCDF file format. Analysis and visualization tools were written to read this file format, so when a climate simulation generates data, that data should be in netCDF format to simplify data management.

Parallel-NetCDF, despite a different API and its parallel I/O features, maintains compatibility with the serial netCDF file format. This compatibility

```

1  ret = ncmpi_create(MPI_COMM_WORLD, filename,
2                    NC_CLOBBER|NC_64BIT_OFFSET, MPI_INFO_NULL, &ncfile);
3  /* after create, file is automatically in define mode */
4  ncmpi_def_dim(ncfile, "d1", DATA_PER_PROC, &(dimarray[1]));
5  ncmpi_def_dim(ncfile, "d2", nprocs, &(dimarray[0]));
6  /* note how this call associates a name, a datatype, and a shape
7   to the variable */
8  ncmpi_def_var(ncfile, "v2d", NC_INT, ndims, dimarray, &varid1);
9
10 ncmpi_def_var(ncfile, "v2d", NC_INT, ndims, dimarray, &varid1);
11
12 ncmpi_enddef(ncfile);
13
14 ncmpi_put_vara_int_all(ncfile, varid1, start, count, data);
15 ncmpi_close(ncfile);

```

FIGURE 15.3: A code fragment demonstrating the two modes of the Parallel-NetCDF API. By requiring a separate declaration step, the data transfer step can occur in parallel without additional coordination.

allowed Parallel-NetCDF to see rapid adoption in the climate community. Science groups could introduce parallelism piecewise, but still use serial tools. For example, the code writing history files in a climate simulation could be updated written in parallel, but the tools to visualize that data could remain serial for a bit longer.

With its long history of using netCDF-formatted files, naturally the climate communities use Parallel-NetCDF. Parallel-NetCDF has also seen adoption in weather, fluid dynamics [5], and astrophysics [3].

Parallel-NetCDF saw tremendous interest from its earliest days. The Leadership Computing Facilities at Argonne and Oak Ridge National Laboratories have installed Parallel-NetCDF for the past three generations of systems. Many other sites have installed Parallel-NetCDF, particularly if those sites host computational scientists from the climate domain. If a user should happen to find Parallel-NetCDF not already installed on a system, its minimal dependencies make building it straightforward.

15.5 Additional Features

Parallel-NetCDF's similarity to serial netCDF provides its strongest selling point. Maintaining that similarity means a new Parallel-NetCDF user need not spend much time learning about the distinction between define mode and data mode or how to view or analyze the generated file. However, Parallel-NetCDF provides several features not available to serial netCDF.

The *Flexible Interface* to Parallel-NetCDF allows a developer to describe arbitrary data in memory when writing to a multi-dimensional file. The various

```

1  /* post two non-blocking operations, writing data to
2  two variables (varid1, varid2) in the \dataset{} */
3  ncmpi_iput_vara(ncfile, varid1, &start, &count, &data, count,
4                 MPI_INT, &requests[0]);
5  ncmpi_iput_vara(ncfile, varid2, &start, &count, &data, count,
6                 MPI_INT, &requests[1]);
7
8  /* here in ncmpi_wait_all the library will inspect all non-blocking
9  operations, combine them, and service the new larger request collectively */
10 ncmpi_wait_all(ncfile, 2, requests, statuses);
11     if (ret != NC_NOERR) handle_error(ret, __LINE__);

```

FIGURE 15.4: A code fragment demonstrating the use of the non-blocking routines. The Parallel-NetCDF library will stitch these two requests into one single more efficient MPI-IO operation.

data transfer methods in serial netCDF (the `var`, `vars`, `vara`, `varm` routines) all take a contiguous memory buffer. Parallel-NetCDF's Flexible Interface allows the caller to specify the type and structure of memory with an MPI datatype. Jianwe Li's SC 2003 paper [4] goes into more detail.

Parallel-NetCDF also introduced non-blocking operations in its API. These nonblocking operations were not interesting at first: they would call the non-blocking MPI-IO routines, but most MPI-IO implementations provided little if any non-blocking support. The non-blocking interface, however, provided a good location to apply an operation-combining optimization.

Figure 15.4 lists a code fragment using this non-blocking interface. Parallel-NetCDF's non-blocking interface follows the same post-and-wait model used in MPI.

Both the Flexible Interface and the non-blocking interface re-enforce a common theme in high performance parallel I/O: providing as much information as possible to the storage system. In Latham et al.'s case study [3], using these extended features of Parallel-NetCDF improved checkpoint bandwidth at scale by a factor of 3.

15.6 Conclusion

The I/O library has seen much research in the decade since Parallel-NetCDF began. HDF5 continues development. ADIOS offers an alternative approach to data management. The serial netCDF project has incorporated parallel I/O features by implementing its API (with some extensions) on top of the HDF5 library. Despite these innovations, Parallel-NetCDF still provides a useful tool in the toolbox of parallel I/O libraries.

The first challenge a scientific application faces when managing data is how to drive the large storage systems in parallel. The developers must first

devise a parallel data decomposition strategy, no matter which library will be used. Often, this decomposition strategy is the trickiest part. Once a strategy is in place, the I/O library used is secondary, and can often be hidden by an abstraction layer (climate codes, for example, use PIO [2] for just this purpose). The glib answer to “Which I/O library should I use?” is “It doesn’t matter, as long as you use *something*.”

Parallel-NetCDF still provides the standard for a low-overhead I/O library. The classic netCDF file format imposes some restrictions, but these restrictions mean the file layout is known ahead of time. The library does not need additional coordination among processes to know where to place data.

Other I/O libraries can borrow ideas proven in Parallel-NetCDF. When serial netCDF introduced parallel I/O to its API, they could use Parallel-NetCDF’s approach for incorporating parallel I/O parameters such as the MPI communicator and how to express collective I/O. An upcoming release from the HDF5 project will contain a “multi-dataset” family of routines following the Parallel-NetCDF approach.

In the flourishing ecosystem of parallel I/O libraries, Parallel-NetCDF still represents a compelling option for codes using regular arrays to describe their data, or are willing to transform their data into such arrays. As scientific data models increase in complexity, the relative simplicity of Parallel-NetCDF might one day no longer be appropriate. As we in the I/O library community develop successor libraries for more sophisticated application data models, Parallel-NetCDF will remain the standard for a lightweight abstraction layer.

15.7 Additional Resources

More Parallel-NetCDF information can be found at the following places:

www.mcs.anl.gov/parallel-netcdf The Parallel-NetCDF home page contains an overview of the package, instructions for joining the mailing list, tutorials, documentation, and bug tracker.

cucis.ece.northwestern.edu/projects/PnetCDF Northwestern University’s Parallel-NetCDF page contains additional material and documentation.

Parallel-NetCDF requires only an MPI-IO implementation, and so is available on nearly every parallel computer. Users should consult local documentation for details of any site-specific quirks.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory (“Argonne”). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government. Please make sure you have the DOE acknowledgment at the end of your paper (before the References).

For acknowledgment information, please use:

This work was supported by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357.